Summer 2018

Developing an energy efficient real-time system

Aamir Aarif Khan

Department:

DEVELOPING AN ENERGY-EFFICIENT REAL-TIME SYSTEM

by

AAMIR AARIF KHAN

A THESIS

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

2018

Approved by

Dr. Maciej Zawodniok, Advisor
Dr. Zhishan Guo, Co-advisor
Dr. Jonathan Kimball

ABSTRACT

Increasing number of battery operated devices creates a need for energy-efficient real-time operating system for such devices. Designing a truly energy-efficient system is a multi-staged effort; this thesis consists of three main tasks that address different aspects of energy efficiency of a real-time system (RTS).

The first chapter introduces an energy-efficient algorithm that alternates processor frequency using DVFS to schedule tasks on cores. Speed profiles is calculated for every task that gives information about how long a task would run for and at what processor speed. We pair tasks with similar speed profiles to give us a resultant merged speed profile that can be efficient scheduled on a cluster. Experiments carried out on ODROID-XU3 are compared with a reference approach that provides energy saving of up to 20%.

The second chapter proposes power-aware techniques to segregate a task set over a heterogeneous platform such that the overall energy consumption is minimized. With the help of calculated speed profiles, second contribution of this work feasibly partitions a given task set into individual sets for a cluster based homogeneous platform. Various heuristics are proposed that are compared against a baseline approach with simulation results.

The final chapter of this thesis focuses on the importance of having an underlying energy-efficient operating system. We discuss an energy-efficient way of porting a real-time operating system(RTOS), QP, over TMS320F28377S along with modifications to make the Operating System(OS) consume minimal energy for its operation.

ACKNOWLEDGMENTS

I would like to express my utmost gratitude to my co-advisor, Dr. Zhishan Guo, for giving me the opportunity to work under him. His immense knowledge and guidance played a vital role in accomplishment of this work. I would also like to thank my advisor, Dr. Maciej Zawodniok, and esteemed member of my committee, Dr. Jonathan Kimball, for providing me wealth of knowledge and support during my time at Missouri S&T.

I would like to dedicate this work to my family, especially my parents without whom I might have never gotten the opportunity to follow my dreams. Their constant support and undying love for me has been my biggest motivation. Lastly, I would like to extend my appreciation to all my friends and everyone else that played a part and helped me through my journey.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

NOMENCLATURE

Symbol	Description
BSP	Board Support Package
CCS	Code Composer Studio
DM	Deadline Monotonic
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
GPIO	General Purpose Input-Output
HMP	Heterogeneous Multi-Processor
HSM	Hierarchical State Machines
ISR	Interrupt Service Routine
LST	Least Slack Time
OS	Operating System
PIC	Prioritized Interrupt Controller
PIE	Peripheral Interrupt Enable
PLL	Phased Locked Loop
QP	Quantum Platform

RM	Rate Monotonic
RTC	Run-to-Completion
RTOS	Real-Time Operating System
RTS	Real-Time System
SoC	System-On-Chip
TI	Texas Instruments

1. INTRODUCTION

Our society has seen tremendous growth in embedded systems in the last few decades. They are efficiently designed that attracts little attention to their presence while serving their purpose, although their impact in the society can hardly be ignored. From electronic calculators to heavy machinery for construction, embedded systems play a major role in the functioning of our society. Its significance has been increasing in the recent years and doesn't show a trend of slowing down in the future. An embedded system is generally designed to have dedicated functions within a bigger electrical or mechanical system to solve specific issues. Unlike a general purpose computer, it does not have monitor or keyboard, but they can be integrated into systems to prompt user interface. They generally encompass at least one microprocessor, both RAM and ROM and some I/O devices. With the advancement of technology, real-time applications in embedded systems are becoming increasingly computational intensive.

1.1. WHAT ARE REAL-TIME SYSTEMS?

A real-time system consists of stringent timing requirements for its applications. They are system that are expected to respond in very short and guaranteed period. real-time applications consists of a task divided into many sub-tasks that can concurrently execute on a processor within the given time-frame. Execution of a task may require access to multiple resources on an embedded platform, for example Gang tasks are tasks that demand simultaneous execution, thus the beginning of those tasks could be delayed as they would wait for a minimum required number of processors to be free such that their parallel execution can be facilitated Feitelson and Rudolph (1992). In other cases, multiple tasks could require access to the same resource like reading/writing to a memory location. A resource can only be used by one task at a time as their simultaneous usage could end up in

garbage values being written or read from that memory location, in worst cases this would prove fatal to the application leading to a system crash. To deal with such situations resource locking/unlocking policies are introduced to prevent cases of deadlock or unlawful usage of resources Sha et al. (1990) Goodenough and Sha (1988) Chen and Lin (1990) Baker (1990). Although a relatively young field compared to many existing fields, real-time scheduling of tasks has been a hugely active research area in academics and industrial applications.

Within a real-time system a correct output is expected within a certain time frame. Failure to complete the task in the expected time could lead to catastrophic results. Let us consider a real-time system designed for an aircraft as an example, the system could be running multiple tasks like measuring air pressure, temperature, velocity etc. all at the same time. The response time for these tasks are generally expected to have very small durations within specified bounds. If the system would fail to achieve them in the expected time slot, it could delay the execution of other tasks. This could very well cause the system to malfunction crashing the plane. Another example we can consider is a real-time system designed to operate a nuclear reactor. As can be imagined, this would require the system to run uncountable calculations every second and with utmost precision. Any sort of delay in the system or an unlawful computational output could very well lead to system instability. This could pose a wide range threat as a nuclear blast would be an absolute disaster causing uncountable loss of property and human lives.

1.2. TYPES OF TASKS

There are three general categories of tasks used in a real-time system viz. periodic tasks, aperiodic tasks and sporadic tasks. Periodic tasks are tasks with constant periods, which means that time between any two consecutive releases of a periodic task is constant. A sporadic task can be understood as a task having a minimum inter-release time between

two consecutive releases. Whereas aperiodic tasks have no relation between its consecutive release times. It can unexpectedly be released at time, the information for which is generally known only during run-time.

From the explained situations we can understand that a real-time system needs careful considerations of a number of factors for rightful functioning. The purpose of a having a real-time system is to make the system predictable, so any abnormal cases can be detected and avoided in early stages of development making the system more robust and reliable. A real-time task has two basic characteristics, one is its logical correctness, the other is temporal correctness. Logical correctness refers to the output of a process, no matter how complex, being logically or mathematically correct. For example a simple addition of $3+4$ should always give an output of 7. Though an easy example in this case, this actual implementation on a processor would be a multi-staged process. The coded program is generally compiled into an executable code that can be stored in the internal memory of a processor. For execution, the processor loads the two variables into two internal registers where values are manipulated in binary. Depending on the coded procedure, the processor performs the computation and gives an output, which would need to be logically correct. If the given output is wrong, the system would fail. Concurrently, the output for a real-time system is expected within a certain duration. For a system running multiple tasks, the execution of one task could lead to delay in completion of another task. A late output is considered as a wrong output in real-time systems, this contributes to its temporal correctness.

1.3. SCHEDULING ALGORITHMS

A real-time system usually assigns priorities to every task that helps the system make scheduling decisions. Priority assignment is decided by the protocol used by the underlying scheduler. At any moment, the scheduler is responsible for choosing the task with highest priority among all available tasks and execute it. There are two main categories of priority

assigning algorithms, one where the priority of every task is pre-assigned and never changes through the course of system execution known as '*static priorities*'. The other is '*dynamic priorities*' where task priorities are assigned during run-time and are changed continuously as execution progresses. Schedulability tests exists for each type of algorithm. They help to ensure whether a a given task set is schedulable or not under a particular algorithm. Some example of static and dynamic priority algorithms are given below.

Static Priority Algorithms. This paradigm consists of algorithms that assigns a permanent priority to every tasks before system start up. Assigned priority can be based on a decided parameter like task period or deadline. *Rate-Monotonic (RM)* algorithm is one of the most famously used algorithms for uniprocessor scheduling. In this algorithm, tasks with shorter periods are assigned higher priorities. This enables the most frequently occurring tasks to be scheduled ahead of the less frequent ones. *Deadline Monotonic (DM)*is a similar algorithm that prioritizes tasks based on their deadlines. Hence tasks with shorter deadlines will have higher priorities. Since the deadlines and periods of a task would not change, neither will their priorities.

Dynamic Priority Algorithm. In contrary to static priorities, dynamic priorities will change a task's priorities every time instant based on the scheduled algorithm. Two of the most widely used dynamic priority algorithms are *Earliest Deadline First(EDF)* where a task's priority at any time instant t is decided based its relative deadline compared to other tasks at that instant. As the name suggests, the task with the earliest deadline will have the greatest priority. *EDF* is an optimal algorithm for scheduling tasks sets on uniprocessors that have total utilization ≤ 1. *Least Laxity First(LLF)* or *Least Slack Time (LST)* is another optimal uniprocessor algorithm that gives higher priority to tasks with lesser laxity. Laxity of a task at any time instant t can be defined as the time left for a job until its deadline. It required the given task set to have a total utilization ≤ 1 as well as requires the system to keep track of more variables compared to *EDF* algorithm.

2. LITERATURE REVIEW

Multi-core platforms are often preferred for applications requiring energy efficiency, performance and real-time guarantees. The paper by Pagani and Chen (2013) show that we can significantly reduce total energy consumption if the load is evenly distributed between two cores rather than assigning the entire load on one core running at double frequency. Much work has aimed at energy-efficient and power-aware scheduling of sequential tasks on multi-core homogeneous systems Bambagini et al. (2016). Chen et al. (2009) and Liu et al. (2012) present an energy-efficient design for scheduling tasks on heterogeneous systems. Problems related with allocating real-time applications in an energy efficient manner onto heterogeneous platforms has been addressed by Colin et al. (2014). Little attention has been given to researching problems relating to power minimization along with intra-task parallelism. Graph tasks are scheduled with minimum power consumption in Zhu et al. (2004) and Zhu et al. (2002). Energy awareness for cores with block partitioning, where cores are divided into blocks sharing a common power supply, was studied by Qi and Zhu (2011). Gang scheduling policy was studied by Paolillo et al. (2014), where a task uses multiple processors in parallel for its execution. Paper published by Chen et al. (2014) considers per-core Dynamic Voltage and Frequency Scaling(DVFS) and introduces a technique to combine dynamic power management with DVFS for dependent tasks. Kong et al. (2011) minimizes energy consumption for tasks with implicit deadlines based on level packing. Non of the work mentioned above considers inter-task processor sharing, the research done in Guo et al. (2017a) is the closest to the work presented in this thesis.

There has been much attention given towards designing underlying operating systems that are focused on conserving power. A number of papers have researched and reported various energy efficient aspects for an Operating System(OS) Lee et al. (1998)Lorch and Smith (1997).Lorch and Smith (1998). The work in Vahdat et al. (2000) focuses on every

aspect from an energy efficient point of view rather than the traditional performance-based approach. Energy is considered as the resource with highest priority managed by the OS in Ellis (1999).

3. ENERGY EFFICIENT REAL-TIME SCHEDULING OF DAGS ON CLUSTERED MULTI-CORE PLATFORMS

3.1. BACKGROUND AND SYSTEM MODEL

Here a brief introduction about real-time systems is given particularly pertaining to aspects relevant to this thesis. Real-time systems can be understood as a system with stringent timing requirements for its task's execution. A real-time task is characterized by dual correctness viz. logical correctness and temporal correctness. That means the achieved output would not only have to be logically correct but also has to be computed within the given timing constraint, else the output not only holds no significance but also could leads to a system failure in worst cases. A system producing a late result is equivalent to one producing an incorrect result. A real-time system generally consists of various tasks with or without any dependency between any two of them. Two kinds of tasks can be considered for a real time system, hard real-time tasks and soft real-time tasks. A task with soft real-time requirements will have steady reduction in usefulness for every timing violation whereas hard real-time constrains are not capable of tolerating any violations of their timing requirements what-so-ever.

3.1.1. Real-Time Task Characteristics. A task τ_i can have the following characteristics:

- Release time, r_i : Time instant at which the task was released
- Worst-case execution time, C_i : Time required to executed a task in the worst case scenario
- Relative deadline, D_i : Duration within which the task needs to complete
- Period, T_i : Duration after which the cycle repeats

They can be written as a tuple (r_i, C_i, T_i, D_i). A task can release many sub-tasks known as jobs. The relation between a task's release and period help us determine its periodicity. A periodic job is released strictly every T time units. A sporadic task however has a minimum inter arrival time between its subsequent releases i.e. they would be released at least T units apart. Finally we have aperiodic jobs which have no predictable release times, they are randomly released by the system. A task with period equal to or more than its deadline ($T \geq D$), is known as a constrained deadline task. If the period is equal to its deadline ($T = D$), it is known as an implicit deadline task.

3.1.2. Directed Acyclic Graph (DAG). A Directed Acyclic Graph (DAG) is a directed graph with finite number of edges and vertices placed in topological ordering as shown in figure 3.1. Each edge is directed from one vertex to another without any cyclic sequence. It would mean that if we started at a vertex A and followed a directed sequence of path, there is no way we can loop back to A. For this thesis, we consider a task set of sporadic tasks denoted by $\tau = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_n\}$, where n denotes the number of tasks in the set and each task τ_i is expressed as a DAG with deadline D_i and minimum inter-arrival separation of T_i time units. Vertexs/Nodes within a DAG represent the execution requirements while edges represent the dependencies among nodes.

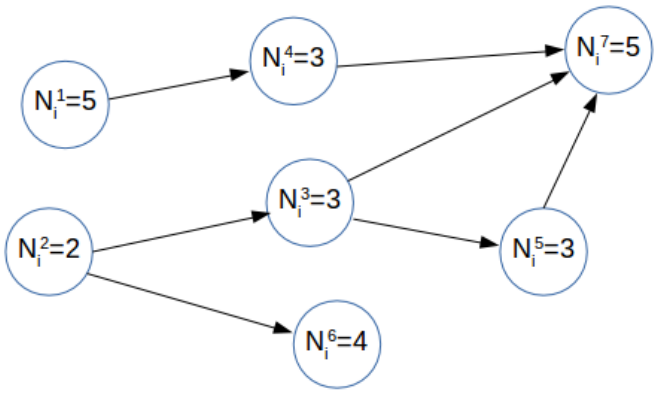

Figure 3.1. DAG representation

A node N_i would be called the parent node or immediate predecessor of node N_k if there exists an edge directed from N_i to N_k. This implies that the execution of N_k cannot start until N_i finishes its execution (predecessor constraint). The execution requirement of node N_i is denoted as c_i^j. Addition of all individual requirements of the nodes of a DAG gives us its total execution requirement denoted by C_i. If we follow a DAG from the start to the end of its graph, the path with longest total execution requirement among all available paths is known as a *critical path*. Sum of execution requirements of all nodes that lie on a critical path is known as the *critical path length*, denoted by L_i. Thus L_i gives us minimum time required to execute a DAG even if we have multiple processors available at our disposal. In turn it is implied that for a task τ_i to be schedulable, at least L_i time units are needed. The above given explanation can be understood better with an example. In the given figure 3.1, the longest path is through nodes $N_i^2 -> N_i^3 -> N_i^5 -> N_i^7$, thus the longest path length will be addition of their individual execution requirements, equal to 15 units.

3.1.3. Power Model. At any given time instant t, we can denote the frequency of a processor as $s(t)$ (we assume continuous frequency scheme for sake of simplicity). Thus the power consumption $P(s)$ of the processor can be calculated by equation 3.1

$$P(s) = P_s + P_d(s) = \beta + \alpha s^\gamma \tag{3.1}$$

In Equation 3.1, $P(s)$ is the power consumption due to leakage current while $P_d(s)$ is introduced due to switching activity. Here $P_d(s)$ is frequency dependent and can be represented as $\beta + \alpha s^\gamma$. $\beta > 0$ is introduced whenever a processor remains on and thus is the static part part of the equation. α is the effective switching capacitance where as $\gamma \in 2, 3$ is a fixed parameter determined by the hardware. The adopted power model is widely accepted in the real-time community Pagani and Chen (2014)Narayana et al. (2016)Huang et al. (2014)Aydin and Yang (2003). Comparison of actual power consumption from Howard et al. and the power model presented in equation 3.1 deemed it to be highly realistic. This work considers a continuous frequency scheme, although our approach is not widely

affected on its application to systems with discrete frequency level. This is owed to the fact that we can get discrete values by rounding up values from the continuous frequency scheme and most state-of-the-art micro-processors available today have a relatively fine grained step-to-scale frequencies. ODROID-XU3 has a frequency range from 100MHz-1400MHz for the LITTLE core and 100MHz-2000MHz for the big core with a scale step of 100MHz. Such fine grained step frequency can still be closely applicable to our approach. Based on this discussion, we can calculate the total energy consumed during interval $[g, h]$ as $E(g, h) = \int_g^h P(s, t)\, dt$.

3.1.4. Platform. The hardware platform we consider for experimentation is ODROID-XU3, a Heterogeneous Multi-Processor(HMP) consisting of Samsung Exynos5422 Octa-core SoC employing ARM's big.LITTLE architecture. Its architecture consists of a 'big' cluster with quad Cortex-A15s and another 'LITTLE' cluster with quad Cortex-A7s. All processors in the same cluster operate at the same frequency as they have the same supply voltage, although the two clusters can operate at their individual frequencies levels. Four TI INA231 power sensors are integrated into the board to accurately measure power consumed by A-7 cores, A-15 cores, RAM and GPU in real-time. The 'big' cluster is the performance cluster whereas 'LITTTLE' is the slower, battery-saving cluster.

3.1.5. Dynamic Voltage and Frequency Scaling. Dynamic Voltage and Frequency Scaling (DVFS) is a technique to change the voltage in a component depending upon circumstances. For situations where a processor might be overloaded with work, increasing the voltage would in turn lead to an increase in the processor frequency. Similarly for lightly loaded work decreasing processor frequency can aid in lesser power consumption. This technique is widely used in mobiles and laptops to reduce power usage and extend battery life. The experiment presented in this thesis utilizes DVFS technique to dynamically alter processor frequency as and when required.

3.2. PROBLEM STATEMENT

We find a feasible strategy to minimize total power consumption for scheduling an implicit deadline and constrained deadline task set on a cluster based system. It is already known that finding an energy efficient partition is NP-hard for both, parallel Li (2012) and sequential tasks Aydin and Yang (2003). Our approach for tacking this problem is as follows: (1) We consider a sporadic task set and apply the existing task decomposition technique followed by the inter-DAG merging from Guo et al. (2017a). (2) With information obtained from the above step, we merge every DAG with a suitable partner such that they both can be assigned on the same cluster while the overall power consumption is minimized.

3.3. PRELIMINARIES

Definition 1 ***(Speed-Profile)*** *When assigned to the cores, the speed-profile of a task describes how long it takes for the task to execute and at what rate of speed. We will represent the speed-profile of a DAG as a random variable $\mathcal{S}$ which has an associated probability function (PF), $f_{\mathcal{S}}(*)$ where $f_{\mathcal{S}}(s) = \mathbb{P}(\mathcal{S} = s)$ and s has a finite set of values. Here, s represents the speed and $f_{\mathcal{S}}(s)$ represents the portion of the DAG period when it is running at this speed (see Example 3.3).*

Example 3.3. Consider a DAG τ_1 with speed profile $\mathcal{S}_1 = \begin{pmatrix} 0.2 & 0.5 \\ 0.3 & 0.7 \end{pmatrix}$, having a time-period of 10 units. The speed profile S_1 indicates that the DAG would run at speed of 0.2 for the initial 3 units of its time period and would run at speed of 0.5 for the rest of its duration i.e 7 units.

3.3.1. Task Decomposition. The *task decomposition* technique breaks down a parallel task τ_i in to smaller individual sub-tasks also known as jobsSaifullah et al. (2014a). This leverages us to schedule them as sequential tasks (preemptive and non-preemptive) along with traditional analysis of multiprocessor scheduling. On decomposition into indi-

vidual tasks, every sub-task has its own release offset, execution time and deadline. These sub-tasks can be scheduled on a multiprocessor forming segments defined by their boundaries i.e. release and deadlines of a sub-task can be understood as the start or end of a segment. Seeming it is a multiprocessor, there can be multiple sub-tasks sharing the same segment as their execution times may overlap. Sub-task release and deadline are assigned in such a way that the original release and deadline of the DAG along with all dependencies between the nodes are respected. Figure 3.2 shows how the DAG represented by figure 3.1 will look like after applying task decomposition technique (Refer Guo et al. (2017a) for more details).

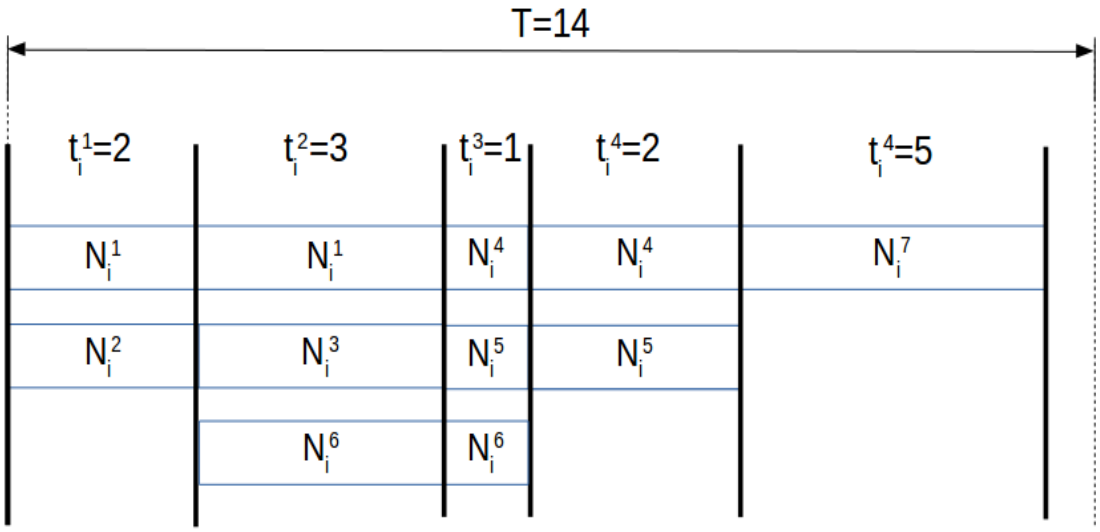

Figure 3.2. DAG after applying task decomposition

3.3.2. Segment Extension. On applying the task decomposition technique we get information like release times and deadline of every node which maybe sufficient but not necessary. This gives room for rearranging the execution of certain nodes to conserve energy. For example consider figure 3.2 where node N_i^6 can be executed in segment 4, allowing us to turn off the third processor, although task decomposition adds an unnecessary condition for it to finish by end of segment 2. Segment extension is a technique to determine the

scheduling window constraints for a node N_i^j. *Scheduling window constraints* refers to the necessary and sufficient time frame beginning from a nodes release offset until its deadline (Refer Guo et al. (2017a) for more details).

3.3.3. Intra-Task Processor Merging. On applying the *task decomposition technique*, we get information about the schedulable windows for individual sub-tasks. On assigning an initial schedule, it may happen that some processors are more heavily loaded than others, such uneven distribution of load may lead to poor energy efficiency. According to Guo et al. (2017a), lightly loaded cores can be combined into one heavily loaded processor. The workload distributed among a pair of lightly loaded cores can be transfered onto just one core, provided it doesn't overflow the total capacity of the processor. During such reallocations, care is taken that no deadline of any DAG is missed. This technique reduces the overall number of cores required, leading to lesser leakage current which is one of the major factors of the total power consumption. We consider a cluster based platform where all cores in a cluster operate at the same frequency, thus it would not be possible to lower down the frequency of a lightly loaded core to save power. However modern microprocessors do have the ability to selectively turn off a processor when not in use. From time-to-time intra-task processor merging may free up a processor which can be turned off, leading to power savings. (Refer Guo et al. (2017a) for more details)

3.4. INTER-TASK MERGING: IMPLICIT DEADLINE TASKS

3.4.1. Choosing Single Speed for the Whole Task Period. Task Decomposition and segment extension and intra-task processor sharing techniques give us various important details like the number of cores required for execution as well as the execution speed of cores at particular instances of time. This information is vital to calculate a single speed for a given DAG. But at any time instant, different nodes within the DAG can require different speeds, so how do we select a single speed at that instant? To tackle this scenario, we select a single necessary speed throughout the task period of a DAG. Motivation for this comes from the

claim derived in Theorem 2 of Guo et al. (2017a): *The total energy consumption (assuming processor remains on) is minimized in any scheduling slot/window when execution speed remains uniform(the same) through the interval.* In our current approach, we determine the workload(execution requirement) among all cores and select the maximum workload from all available ones to calculate the aggregate workload. The desired single speed for a DAG is achieved by dividing the aggregate workload by the task period. Consider a task τ_i allocated to M cores. For any segment j having a length of t_j^c, the workload and speed of a core k can be given by $w_{i,j,k}$ and $s_{i,j,k}$. Workload can be calculated as,

$$w_{i,j,k} = s_{i,j,k} \times t_j^c.$$

The maximum workload per segment among all cores is calculated by,

$$w_{i,j} = max(\forall_k(w_{i,j,k})).$$

The desired single speed and the aggregate workload w_i for a task can be calculated using the following equation,

$$w_i = \sum_{j=1}^{Z} w_{i,j}, \quad s_i = \frac{w_i}{T_i}. \tag{3.2}$$

Here, Z denotes the total number of segments in τ_i. A single speed is calculated for the whole task period and denoted by P, which is represented as $< s_i, p_i >$. Here, the probability of a cluster to run at speed s_i will be p_i.

3.4.2. Greedy Merging. In subsection 3.4.1 we introduced algorithm 1 that outputs a single executable speed s_i for every DAG τ_i. This section explains the technique to find the most suitable DAGs for merging based on their calculated single speeds and assign them to be scheduled on the same cluster. We follow a greedy approach for selecting the most suitable pair, the proposed steps for inter-DAG processor merging are as follows:

Algorithm 1: Single Speed for a Task

Input: Speed $s_{i,j}$ at every segment j, for a task τ_i.
Output: A single speed s_i for the whole task period.
$w_i = 0$; ▷ total workload of τ_i;
for $j = 1$ to total Segments **do**
  $maxLoad = 0$;
  for $k = 1$ to total Cores-1 **do**
    $w_{i,j,k} = s_{i,j,k} \times t_j^c$;
    $maxLoad = max(maxLoad, w_{i,j,k}, w_{i,j,k+1})$;
  end
  $w_{i,j} = maxLoad$;
  $w_i = w_i + w_{i,j}$;
end
$s_i = w_i/T_i$;
return s_i;

1. Initially, all speeds are marked unselected.
2. Stat with the largest speed and mark it selected.
3. Start from the largest unselected speed and try to merge it with one selected in step 2.
4. Calculate power savings according to the merging technique discussed in Subsection 4.1 of Guo et al. (2017a), merge them into the same cluster with speed of the one selected in step 2 and mark it selected as well.
5. Follow instruction given in step 3 until no more speeds can be merged.
6. Follow instruction given in step 2 until all speeds are selected.

3.5. INTER-TASK MERGING: CONSTRAINED DEADLINES TASK

As explained before, tasks with constrained deadlines have their deadlines shorter than their periods. This makes a task more heavily loaded compared to the case with implicit deadline and imposes tighter constraints. We propose two different approaches in this section for creating a DAG's speed profile and also discuss greedy pairwise merging of tasks for maximum power savings.

3.5.1. Creating Speed-Profile. To create the speed-profile of a DAG, we propose two approaches, *(1) Choose the maximum speed among all running cores within a cluster for any time instant t and (2) calculate single speed for the whole task deadline*.

Maximum speed at each segment. As mentioned before, by applying the task decomposition and segment extension technique we can get important information for a DAG like duration and execution speed of each segment and the number of cores required to schedule the DAG. For this approach, we consider the maximum speed of a core among all available cores within a cluster at any time instant t(see Algorithm 2). This ensures that we always operate the cluster at a speed that can satisfy execution of even the most heaviest node. For tasks with *constrained deadlines*, its execution has to be completed by deadline D, where $D_i \leq T_i$. Thus for the rest of the time$(T_i - D_i)$, we can assume the core is idle. We create a pair P_j for every segment $j \in \tau_i$, where $P_j =< s_{i,j}, p_{i,j} >$. Here, $s_{i,j}$ denotes the maximum speed of the j^{th} segment and $p_{i,j}$ denotes the probability of the cluster running on that speed. For any given segment j, the maximum speed $s_{i,j}$ can be calculated using as,

$$s_{i,j} = max(\forall_{k \in M}(s_{i,j,k})).$$

Here, M denotes the number of cores allocated to task τ_i. Its probability cab be calculated as,

$$p_{i,j} = \frac{t_j^c}{T_i}.$$

The expected speed profile S_i will look like so,

$$\mathcal{S}_i = \begin{pmatrix} s_{i,1} & s_{i,2} & \cdots & s_{i,z} \\ p_{i,1} & p_{i,2} & \cdots & p_{i,z} \end{pmatrix}.$$

As the cluster will remain idle for duration $(T_i - D_i)$, we will add an additional pair P_{j+1}, where $P_{j+1} =< 0, (T_i - D_i)/T_i >$. Thus, S_i will be,

$$\mathcal{S}_i = \begin{pmatrix} s_{i,1} & s_{i,2} & \cdots & s_{i,z} & 0 \\ p_{i,1} & p_{i,2} & \cdots & p_{i,z} & (T_i - D_i)/T_i \end{pmatrix}.$$

Algorithm 2: Max Speed at Each Segment

Input: A task τ_i, with speed $s_{i,j}$ at each segment j.
Output: Maximum speed at each segment.
for $j = 1$ to total Segments **do**
    $maxSpeed = 0$;
    for $k = 1$ to total Cores-1 **do**
        $maxSpeed = max(maxSpeed, s_{i,j,k}, s_{i,j,k+1})$;
    end
    $s_{i,j} = maxSpeed$;
end
return $s_{i,j}$;

Example 3.5.1 Consider we have an implicit deadline task with deadline as 9 and time period as $12(D_i = 9, T_i = 12)$ divided into two segments. We consider the maximum speed at each segment, thus there will be two pairs P_j each expressed as $< s_{i,j}, p_{i,j} >$. Since the give deadline is 9, the sum of all time segments should be 9.

Let us consider that $t_1^c = 2.5$, $t_2^c = 4$, $s_{i,1} = 0.55$ and $s_2^c = 0.78$. Probability Values for creating the speed-profile can be calculated as $p_1 = 2.5/12 = 0.21$ and $p_2 = 4/12 = 0.3$. Since the task in consideration is an implicit deadline task, where $(D \leq T)$, we will add an addition of speed 0 for the idle period, $p_3 = 0.54$.

Thus final speed profile will P_j will be,

$$\mathcal{S}_i = \begin{pmatrix} 0.55 & 0.78 & 0 \\ 0.21 & 0.3 & 0.49 \end{pmatrix}.$$

Single Speed Throughout. Another technique we propose is to calculate a single speed for the DAG. The algorithm is similar to that followed in section 3.4.1, except for a minor change. Since we are considering constrained deadlines, we calculate the single speed for the whole deadline rather than for the whole period. Thus the workload is divided by D_i instead of T_i. Since we consider single speed, the speed profile of a DAG would now consist of one pair $< s_{i,j}, p_{i,j} >$ and another pair with speed zero added for the idle period.

3.5.2. Task Combination:Greedy Merging. In this section, we introduce the method to merge two tasks with similar speed profiles for efficient energy consumption during their execution. But in order to understand this technique, some preliminaries are explained as given below:

The previous sections give us the methods to decompose a DAG into individual sub-tasks and schedule them efficiently on a multiprocessor such that the all deadline are respected and we have minimum cores required to schedule the DAG. This section explains the technique to merge a DAG with its best fit pair to efficiently utilize a processor for their execution. We consider a sporadic task set $T_s = \{\tau_1, \tau_2, ...\tau_n\}$ with n tasks, where $(1 \leq i \leq n)$ and every task τ_i is represented as a DAG. In a clustered multi-core system, the number of cores per cluster is fixed and at any given time, all cores in a cluster operate at the same frequency. Due to the sporadic nature of a task and the underlying requirement of a clustered multi-core platform, it is extremely difficult to predict the exact speed of a cluster at any given instant. To tackle this issue, the operating speed of a cluster at any given time is calculated by the *probabilistic speeds-profiles* of tasks. The speed-profile of a task indicates the duration of how long a task would be executed for and at what speed. We can use the speed-profiles of two DAGs and merge them to create an resultant speed-profile that satisfies the execution of both the DAGs and gives us the speed to run the cluster on.

Example 3.5.2 Figure 3.3 gives a visual representation of how speed selection for a cluster can be affected by the sporadic arrival of tasks. In the given example, we have two sporadic tasks τ_1 and τ_2 each with their own speed profile as shown in figure 3.3. Consider

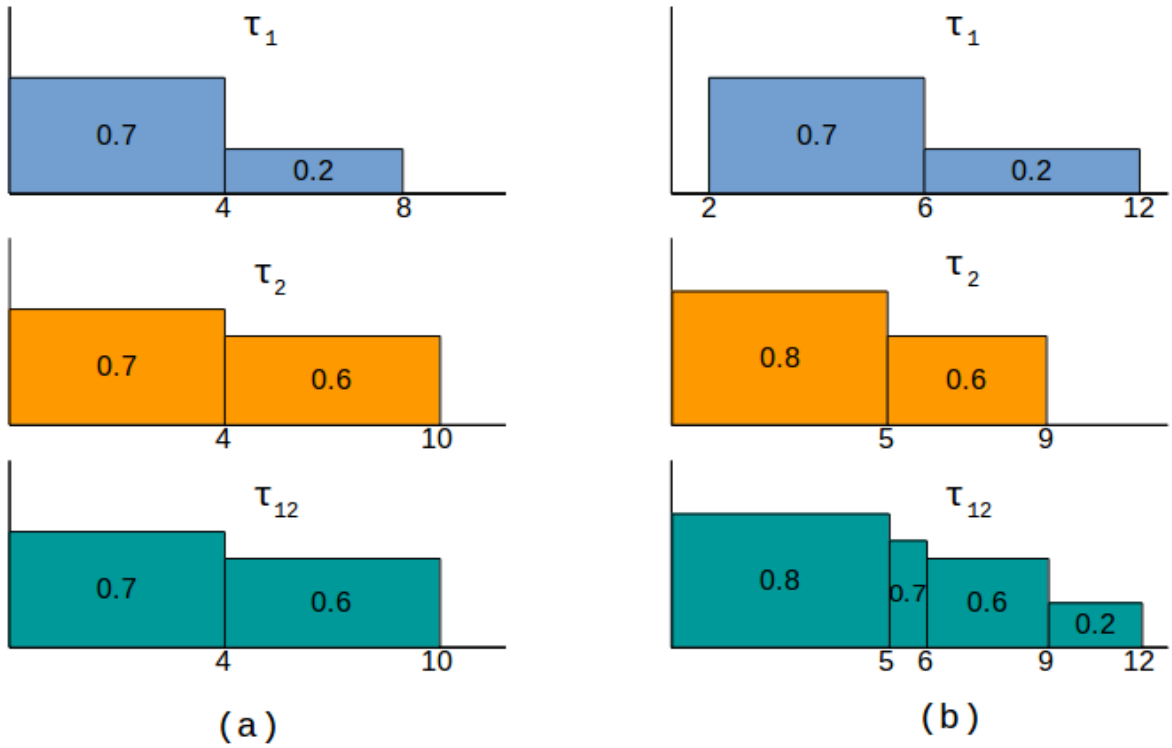

Figure 3.3. Two DAG tasks τ_1 and τ_2 with different speeds and arrival times and a resultant merged DAG τ_{12} with resultant speed pattern. Values closed in rectangles denote the execution speed. X and Y axis denote the time and speed respectively.

figure 3.3(a) where the periods of the two DAGs are 8 and 10 time units respectively. As all cores within the same cluster run at the same speed for any given time instant t, speed of the cluster for time interval [0-4] will be 0.7 units. For interval [4-8], τ_1 executes at speed 0.2 units and τ_2 executes at speed 0.6 units. In order to ensure that both DAGs meet their deadlines, the cluster should execute at a speed of 0.6 units during that time interval [4-8]. For the remaining time interval [8-10], the cluster will operate at the speed of 0.6 units.

Consider figure 3.3(b) where the tasks arrive at different times. Here the tasks have a relatively different speed profiles as compared to figure 3.3(a). Since we need to maintain the maximum cluster speed required by any task for at any time instant t, the cluster would

execute at a speed of 0.8 units for the interval [0-5] and change to 0.7 units for interval [5-6]. Similarly, speed of the cluster will be 0.6 units for time interval [6-9] and will be 0.2 units for the rest of the interval [9-12].

For two DAGs that have overlapping execution times, the operating frequency for the give cluster at any time instant would be the maximum speed required among both the DAGs at that instant. Doing so ensures two primary objectives viz. all the cores within a cluster operate at the same frequency and deadlines of both the tasks are met. Selecting the maximum speed ensures that we satisfy the need of even the heaviest task at any given moment, thus not violating its deadline. To do this, we introduce a special operator, $\odot$. It operates on two given variables and returns the larger one.

Definition 2 *Give two variables $\mathcal{X}$ and $\mathcal{Y}$, special operator $\odot$ performs an operation on both the variables and returns the larger one. During this operation, each entry $\mathcal{X}_i$ ($\mathcal{X}_i \in \mathcal{X}$) is compared with each entry $\mathcal{Y}_i$ ($\mathcal{Y}_i \in \mathcal{Y}$) and calculates $\mathcal{Z}_{ii}$ as $\mathcal{Z}_{ii} = max(\mathcal{X}_i, \mathcal{Y}_i)$. It multiplies the probabilities associated with $\mathcal{X}_i$ and $\mathcal{Y}_i$. Lastly, multiple entries of the same speed values are merged into a single entry with their associated probabilistic values summer together.*

Example 3.5.2 Let $\mathcal{X} = \begin{pmatrix} 7 & 3 \\ 0.2 & 0.8 \end{pmatrix}$ and $\mathcal{Y} = \begin{pmatrix} 7 & 5 \\ 0.2 & 0.8 \end{pmatrix}$.

Then $\mathcal{Z} = \mathcal{X} \odot \mathcal{Y} = \begin{pmatrix} 7 & 7 & 7 & 5 \\ 0.04 & 0.16 & 0.16 & 0.64 \end{pmatrix} = \begin{pmatrix} 7 & 5 \\ 0.36 & 0.64 \end{pmatrix}$.

When $\mathcal{X}_2$ (value 3 with probability 0.8) is compared with $\mathcal{Y}_2$ (value 5 with probability 0.6) $\mathcal{Z}_{22}$ becomes 5 ($\max(\mathcal{X}_2, \mathcal{Y}_2) = 5$) with probability 0.64. Finally, the repeated values ($\mathcal{Z}_{11}$, $\mathcal{Z}_{12}$, and $\mathcal{Z}_{21}$ in this example), are merged into a single entry while their probabilities are summer together.

Consider two task τ_i and τ_j, we can calculate the respective speed profiles $\mathcal{S}_i$ and $\mathcal{S}_j$ by following steps given in subsection 3.4.1. The method to calculate a resultant speed profile by merging both tasks is illustrated in Example 3.5.2. To choose two tasks for merging that would share the same cluster, we greedily choose the pair that provides maximum power savings according to section 4.1 of Guo et al. (2017a). It should be noted that their approach cannot be directly applied to our case. Work done in Guo et al. (2017a) merges cores within the same DAG, whereas we merge two DAGs that will be allocated on the same cluster. We use our concept of *speed-profiles* to tackle this problem. The profile gives us information about the speeds required by a task and their probabilistic values within the task's period, thus we do not need to consider the period as the values are probabilistic ones. Another difference to be noted is that Guo et al. (2017a) simply sums up the speeds of two cores during merging. In our case, we take consider the maximum execution speed at any given time instant *t*.

We allow merging of two DAGs that previously haven't been merged before. The *pseudo-code* presented in algorithm 3 elaborates represents the above mentioned steps. We begin with two empty lists $\bar{\mathcal{S}}$ and $\tilde{\mathcal{S}}$ that will hold the possible and selected speed profiles. Lines 2 – 6 calculate the temporary possible speed profiles and insert them into $\bar{\mathcal{S}}$. The pair of DAG providing maximum energy savings is selected greedily and put into $\tilde{\mathcal{S}}$. We update $\bar{\mathcal{S}}$ by removing the selected pair, preventing it from further merging. The final list $\tilde{\mathcal{S}}$ is returned.

3.6. EXPERIMENT

This section elaborates the details and procedures carried out for experimentation on the ODROID-XU3 board. ODROID-XU3 is a powerful and energy efficient computing device. It can be run as a stand-alone computer with open source support offering various operating system choices including Ubuntu 16.04, Android 4.4 KitKat as well as 7.1 Nougat. To support advanced processing on ARM devices, it implements eMMC 5.0, Gigabit

Algorithm 3: Greedy Merging

Input: Task-set τ, with speed-profile $\mathcal{S}_i$ for each task.
Output: Speed-profile $\tilde{\mathcal{S}}$ (with processor power saving).
$\bar{\mathcal{S}}, \tilde{\mathcal{S}} \leftarrow \emptyset$ ▷ All the possible/selected speed-profiles;
for $i = 1$ to n **do**
  for $j = i + 1$ to n **do**
    $\mathcal{S}_{ij} \leftarrow \mathcal{S}_i \odot \mathcal{S}_j$; $\bar{\mathcal{S}} \leftarrow \bar{\mathcal{S}} \cup \mathcal{S}_{ij}$;
  end
end
while $\exists \mathcal{S}_{xy} \in \bar{\mathcal{S}}$ and $\mathcal{S}_{xy}$ provides non-zero power saving **do**
  $\mathcal{S}_{xy} \leftarrow$ the pair from $\bar{\mathcal{S}}$ with maximum power saving;
  $\tilde{\mathcal{S}} \leftarrow \tilde{\mathcal{S}} \cup \mathcal{S}_{xy}$;
  for $k = 1$ to n **do**
    $\bar{\mathcal{S}} \leftarrow \bar{\mathcal{S}} - \mathcal{S}_{kx} - \mathcal{S}_{ky}$;
  end
end
return $\tilde{S}$;

Ethernet Interfaces and USB 3.0 which boasts amazing data transfer speeds. As mentioned before, ODROID-XU3 employs ARM's big.LITTLE architecture with two cluster islands. Four integrated TI INA231 power sensors provide us with accurate and real-time power reading for various components on the board like A-15 cluster and A-7 cluster GPU and RAM. For the experiment, we directly utilize on-board sensors to get accurate power consumption while an energy monitoring script, emoxy3 Energy Monitoring logs the energy consumption of the workload.

3.6.1. DAG Generation. Our workload is generated using the Erdös-Rényi method-Cordeiro et al. (2010). It is as well know method for generating DAG task sets. For any given number of nodes n in a DAG, the probability of having a connection between two nodes in represented by p. This method does not guarantee to generate to produce a connected DAG. Hence in case a disconnected DAG is generated, we append the fewest number of edges required to make the DAG connected. In our case, we set p to 0.25. For fixing task periods to our set, we consider arbitrary periods, where every period T_i is determined using a Gamma distribution Gamma distribution and set T_i as, $T_i = L_i + 2(C_i/m)(1 + \Gamma(2, 1)/4)$

Saifullah et al. (2014b), Guo et al. (2017b), where critical path of task T_i is denoted by L_i. We compute the critical path length for every DAG as mentioned in section 3.2 i.e. sum of execution requirements of all nodes that lie on the critical path for a DAG.

To better demonstrate the effectiveness of our proposed algorithm, we generate a two task sets of 300 DAGs each. Keeping in mind the architectural nature of our evaluation platform, we assign one set of tasks to run on the big core while the other set is for the LITTLE core. Energy consumption over a period of 230000ms was measured, this is one hyperperiod of the DAGs.

3.6.2. Rt-App. The most basic unit of execution in a processor is known as a thread. DAGs and its nodes are represented as threads in the system. Their generation and workflow is achieved by using the POSIX Thread model, also known as *pthreads*. POSIX Threads is an execution model that allows control over multiple different workflows that may have timing overlaps. It is an API defined by standard POSIX.1c, (IEEE Std 1003.1c-1995). It is freely available bundled in many Unix-like operating systems such as NetBSD, Linux, Mac OS X, Android, FreeBDS and Solaris, typically as '*libpthread*' library. For more information, please refer to Rt-App documentation.

Calls to *pthread* API are managed through '*rt-app*' program. *rt-app* is one of the scheduler tool available in Linux, typically used to emulate real-time system use cases along with giving their runtime information. Through *rt-app* we generate the workload for each DAG which utilizes the POSIX Thread model to call and execute threads. The life cycle of these threads are bounded by execution time, period, priority, core assignment along with other values that can be specified with *rt-app*. *rt-app* accepts a JASON file as an input that defines these aforementioned thread values. In this experiment, we randomly select the execution time for each node to be between [300ms, 700ms]. Please note that *rt-app* itself occurs a certain varying latency between 13-150ms every time it is called, hence we add the upper limit of this i.e. 150ms to the execution time of every thread.

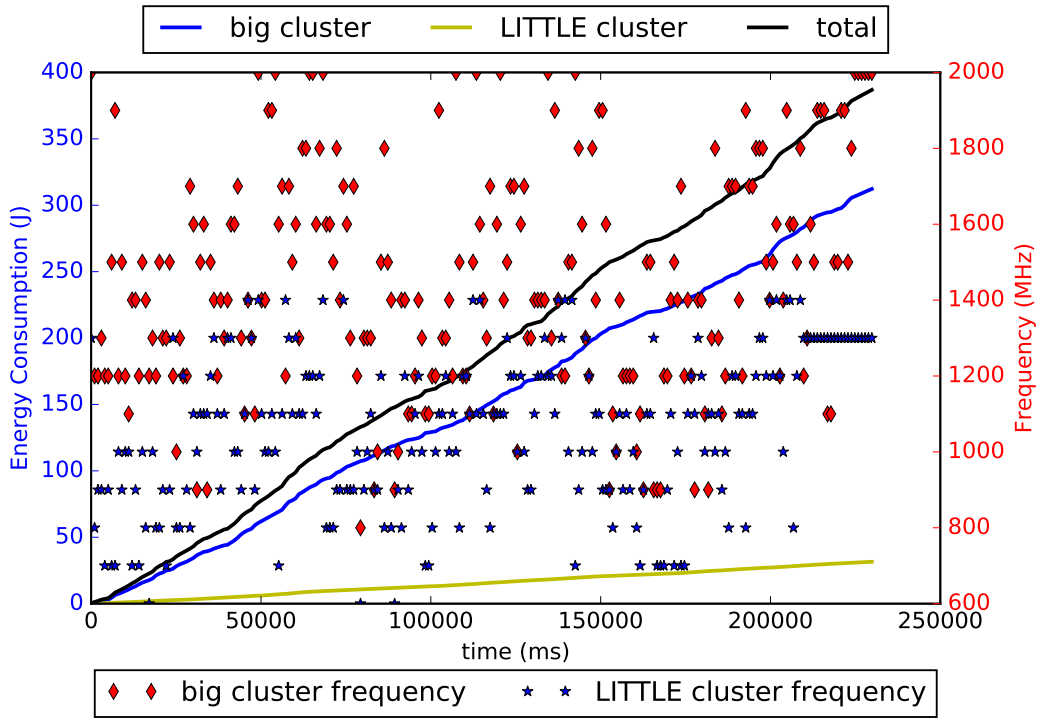

Figure 3.4. The energy consumption and frequency variation of our proposed approach on ODROID-XU3

3.6.3. DAG Scheduling. DAG scheduling is carried out by using the Linux built-in scheduler *sched_FIFO*. *sched_FIFO* is a built-in scheduler in Linus that implements a fixed-priority scheduling algorithm. All DAG tasks have been given a priority higher than other system tasks to ensure no system task interferes with our DAG execution. It should be noted that our approach is also applicable to other work-conserving scheduling algorithms.

3.6.4. Frequency Scaling. We deploy a run-time monitor that detects the arrival and completion of a nodes in the system. Based on the frequency/speed profile for a particular node as mentioned in section 3.4, system frequency is scaled using *cpufreq-set* program from the *cpufrequtils* package. The main overhead in our experiment is due to the online frequency scaling mechanism. As measured on ODROID-XU3, the big cluster takes at most 40 ms to 60 ms for scaling-down and scaling-up respectively. Whereas on the LITTLE cluster, it takes at most 15ms for scaling both up and down.

3.6.5. The Reference Approach. No previous work has studied the problem we address in this thesis, hence we did not directly find a proper reference approach for comparison form literature. We consider a reference approach that studied energy efficient scheduling of sequential tasks in Chen and Kuo (2007), where every task is assigned a frequency it operates at and scheduled at run-time based on their individual operational frequency. For the reference approach, we compute the operational frequency for each DAG. While ensuring all deadlines of all DAGs, their execution times are stretched out by the operational frequency as much as possible. For fair comparison, the reference approach, like our approach, also executes two sets of DAG, one for the big cluster and the other for the LITTLE cluster without the merging technique proposed in section 3.4.

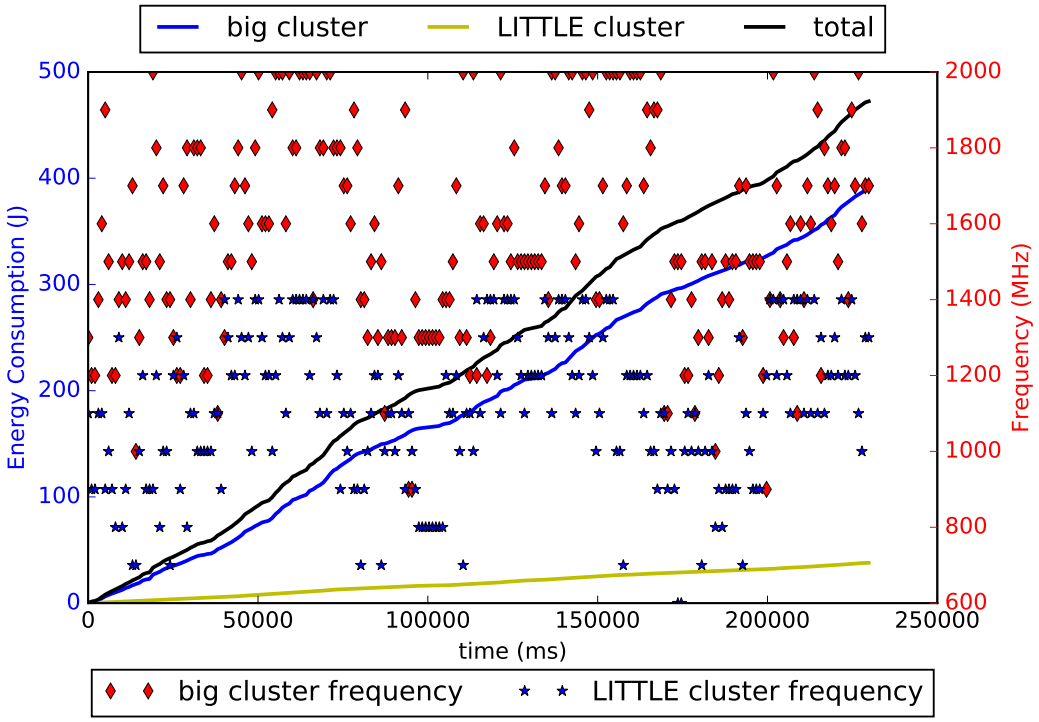

Figure 3.5. The energy consumption and frequency variation of the reference approach on ODROID-XU3

3.6.6. Results. Experimental results are plotted in Figure 3.4 and 3.5. In the given figures we display (1) operating frequencies of the big and LITTLE cluster and (2) energy consumption over the hyperperiod interval for all DAGs i.e. 230000ms. The three lines in

the figure give us the energy consumption of the big cluster, LITTLE cluster and the total system system consumption. It can be observed that the total energy consumption is higher than the summation of big and LITTLE cluster energy consumption. This is owing to the fact the the total system consumption also includes energy consumed by GPU and DRAM, however there is negligible difference between the two approaches for GPU and DRAM consumption. Operating frequency levels of the big and LITTLE cluster are denoted by diamonds and stars at a particular instant, respectively.

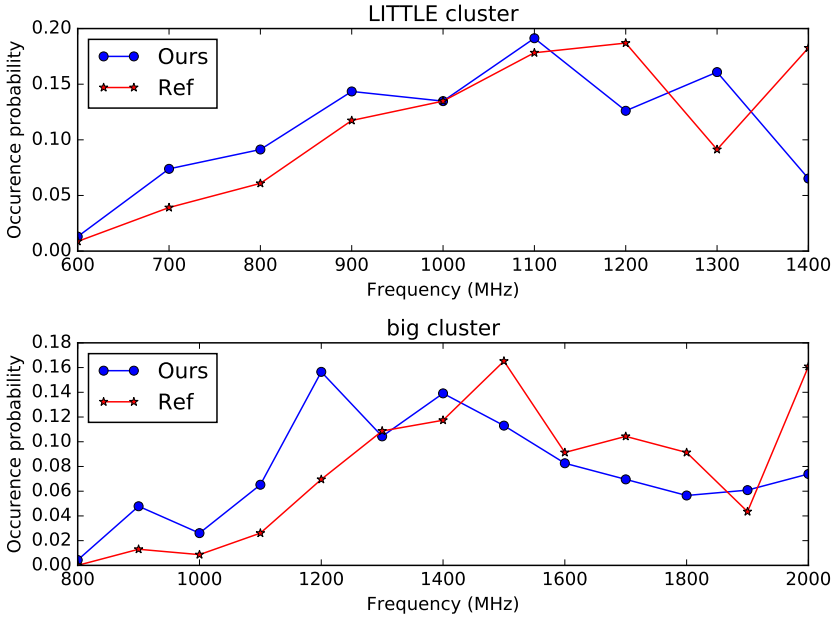

Figure 3.6. Frequency occurrence probability

Comparison of results for both approaches are summarized in table 3.1, where energy consumption for both clusters along with the overall system energy consumption and the energy saving by our approach is presented. As seen in the table, our approach consumes 32 J and 312 J on the LITTLE and big clusters, respectively. In comparison with the reference approach, our approach saves energy by 20% and 16% on the clusters, whereas it saved 18% total energy consumption.

	Ours (J)	Ref (J)	Energy Saving (%)
big cluster	312	389	20
LITTLE cluster	32	38	16
Total	387	472	18

Table 3.1. Summary of experimental results

The reference approach only scales the system frequency per DAG, whereas our approach can change the frequency during the scheduling of a DAG where ever applicable, thus giving us the advantage of having a much finer grained frequency scaling. As mentioned before, the operational frequency is recorded by emoxu3 every 1000 ms. Figure 3.6 shows the probability of a frequency occurring on the big and the LITTLE cluster. It can be seen in the figure that for a given time interval, the reference approach has a much higher probability of executing at high frequency, specially at the max frequency, thus leading to higher energy consumption. Our approach on the other hand has a greater chance of executing at a lower frequencies, thus leading to a lower energy consumption.

4. TASK-TO-CORE MAPPING ON HETEROGENEOUS PLATFORM

The previous section introduced an energy efficient algorithm to schedule DAGs on their assigned cluster. In this section, we propose techniques that can be used to map a given task set on a heterogeneous platform.

For ease of explanation, rest of the paper will focus on application of our theory to ARM's big.LITTLE heterogeneous architecture but the concept itself is certainly applicable to other heterogeneous platforms. A given task set is scheduled into two task sets, one for the big cluster, the other for LITTLE cluster, such that when scheduled, the total power consumption from both cores will be minimal. In this section we explain how a task's speed-profile can be used to estimate its power consumption by using our power model.

Let us consider the power equation introduced in section 3.1.3, where the power $P(s)$ consumed by a processor running at a frequency $s(t)$ at any time instant t can be calculated as,

$$P(s) = P_s + P_d(s) = \beta + \alpha s^{\gamma}$$

Also, speed-profile of a task can have a general expression of $\mathcal{S} = \begin{pmatrix} s_1 & s_2 \dots s_e \\ p_1 & p_2 \dots p_e \end{pmatrix}$, where e depicts the maximum number of columns a DAG contains in its profile.

To understand the correlation of the speed-profile and the power model, we once again direct our readers' attention to work published by Guo et al. (2017a), particularly to Section 4.1, equation (8). It gives a direct relation between a task's speed profile and its power consumption on a processor running at speed s. The equation is as given below,

$$P_j = \beta + \alpha \sum_k \frac{t_i^k}{T_i} (S_j^k)^{\gamma} \tag{4.1}$$

P_j represents the power consumed by the k^{th} portion of a task τ_i running with speed S_j^k on the j^{th} processor. For example, consider the speed profile S_l for a DAG task τ_i with deadline 30 running with speed 0.2 for initial 12 units of time and speed 0.4 for the latter portion on the LITTLE core can be given as, $\mathcal{S}_l = \begin{pmatrix} 0.2 & 0.4 \\ 0.4 & 0.6 \end{pmatrix}$.

the power consumed on one core can be calculated as,

$$P = \beta + \alpha(\frac{12}{30}.0.2^{\gamma} + \frac{18}{30}.0.4^{\gamma})$$

As mentioned before, α, β and γ are hardware dependent values which can be found in a work published by Liu et al. Liu et al. (2015). Knowing the value of α, β and γ for the big and LITTLE core and a task's speed-profile, we can calculate the power required by a task for both cores.

4.1. CHALLENGES

Depending on the core assigned, a task can have varying energy consumption. The processor-speed values for a tasks' speed profile will be larger for the power intensive core and will be lower for the energy saving core. The segregational duration would stay the same for both as they represent the part of the tasks' time-period during which the DAG may have greater or lower speed. Task-to-Core mapping is a well known NP-Hard problem. The overall power consumption also depends on hardware specific variables α, β and γ as we have seen in the problem statement, which would change if our target platform changes. The total utilization per cluster is bounded which prevents us from infinitely allocating tasks to cores. As the number of cores are limited, so should be the total utilization of allocated tasks.

4.2. HEURISTICS

To tackle the challenges mentioned in Challenges, we propose heuristics that provide energy-efficient mapping of tasks to ARM's big.LITTLE core and compare the results. Our solution would partition the original task set into two resultant task sets, one containing all tasks assigned to the big cluster and the other containing tasks assigned to the LITTLE cluster. For all heuristics, we will assume that our initial task set is schedulable with total utilization equal to double of max utilization of the LITTLE core. This helps us guarantee the schedulability of our resultant task sets as some tasks will be alloted to the big core, where the utilization will always be within bounds.

4.2.1. Greedy Algorithm. Greedy algorithm is a well know algorithmic paradigm for solving NP-hard problems, offering various algorithms, each with its own advantages and disadvantages. Some examples of the most widely used greedy algorithms are First Fit, Best Fit, Worst Fit Decreasing/Increasing etc. These algorithms hope to find the globally optimum solution by making locally optimum choices.

First Fit. We resort to implementing the First Fit Algorithm to map our given task set on the two clusters. For the given task set, choosing tasks in a First-Come-First-Serve basis, we find the first cluster that can schedule the given task with minimum power consumption without violating its utilization constrains. On finding a suitable cluster, the chosen task is assigned to the big or LITTLE task set, respective. This is repeated until all tasks in the original task set are assigned to either a big or the LITTLE core.

4.2.2. Randomization. Randomization is known to be another effective method to tackle NP-hard tasks, often achieving better results that greedy approaches. As the name suggests, all tasks are randomly assigned to the big and LITTLE cluster. The random assignment is repeated multiple times and the power consumption for each assignment is recorded.

4.2.3. Non-Linear Programming. Non-Linear Programming(NLP) consists of tackling problems with non-linear difficulties. We tackle our task our minimization problem to solve it using MINLP with constrained variables, where our solution is $(1 \times n)$ vector consisting of binary values. Here n represented the number of tasks in our initial task set where a 1 in the solution vector would represent that the task is allocated to the big cluster and a 0 would represent that the task is allocated to the LITTLE cluster.

4.2.4. Genetic Algorithm. Genetic Algorithm is a technique that attempts to solve problems by replicating biological evolution. Its a natural selection based process that can be used to solve problems with constrained as well as non-constrained variables. The algorithm usually begins with an initial *population*, which can be a set of random solutions for the given problem. In our case, for a task set with n tasks, the initial *population* can consists of a random $(1 \times n)$ binary vector with 1 representing that the task is alloted to the big cluster and a 0 meaning it is alloted to the LITTLE cluster. Here, every element in a solution set can be understood as *gene*, many genes combine to form a chromosome. A bunch of chromosomes together are called a population. With the help of a fitness function a score is assigned to each individual(solution). This fitness score also decides the probability of an individual to be chosen for cross-over, individuals with higher fitness scores are more likely to be chosen that those with lower ones. On choosing a pair of individuals based on their fitness scores, a cross-over point is selected at random and the genes are interchanged among parents until the cross-over point, this gives rise to an offspring which are added to the population. A pre-set probability changes the individual genes within certain offsprings to maintain diversity within the population and prevent premature convergence. This process is repeated until the there isn't significant difference between the new offsprings being produced and its parent population.

4.2.5. Baseline: Brute Force Method. Brute force method refers to trying every possible combination in a give set. Though effective it is not generally preferred as its complexity is $O(2^n)$, where n denotes the total number of tasks. We consider this as our baseline as it could give us the least power consumed and optimally partition the set into two sets.

4.3. RESULTS

Simulation results carried out on MATLAB for allocating a tasks set on ARM's big.LITTLE architecture are presented here. A task set consisting of 20 tasks with initial utilization equal to maximum utilization for the LITTLE cluster. Since LITTLE cluster consists of 4 cores, our initial utilization is 4. Maximum utilization of the task set is incremented by 0.05 at every iteration until maximum utilization is equal to sum of big cluster utilization and LITTLE cluster utilization. This procedure is carried out for every technique presented in section 4.2.

Figure 4.1 shows the plotted values of minimum power consumed in Watts calculated by the different techniques proposed, brute force method acts as our baseline.

4.3.1. Brute Force. We calculate the minimum power consumed during each iteration by trying all possible allocation for the tasks in the task set. In our case the task set consists of 20 tasks, thus we can schedule the task set 2^n ways on the big and the LITTLE cluster. Each combination is checked whether it satisfies the utilization constraints for both clusters, if it does, the power consumption for that combination is noted. This increase the computation time required as the number of tasks increase. We can notice in figure 4.2, the time complexity increases exponentially as the number of tasks increases. We keep a track of the least power consumed through all iterations. Brute force power consumption is represented by the blue line in figure 4.1, it almost perfectly overlaps with power consumption by the genetic algorithm.

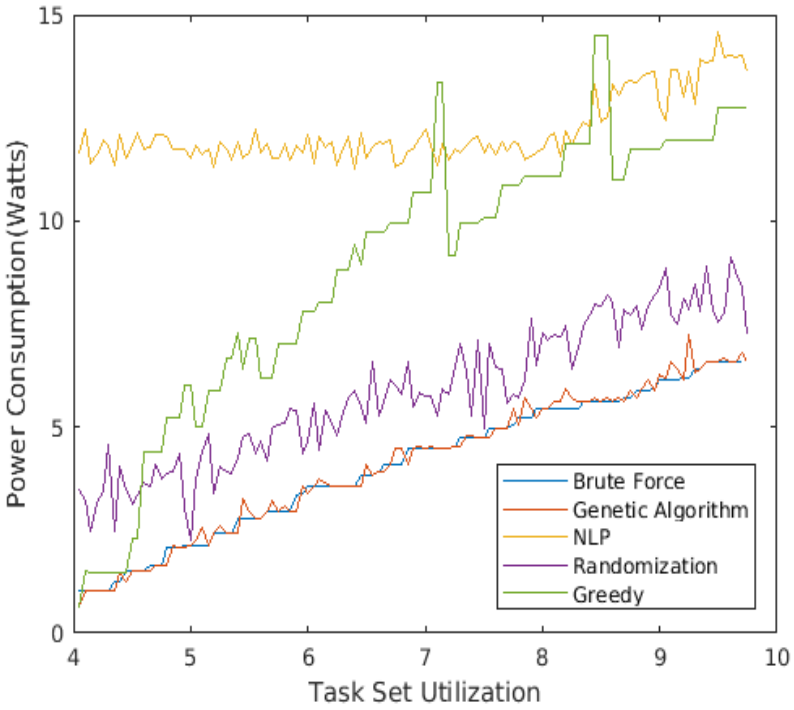

Figure 4.1. Variation in power consumption of taskset with 20 tasks as utilization increases

4.3.2. Greedy Algorithm. Greedy algorithm aims at finding the first available core consuming least power and thus it often tends to get stuck in a local minima. Hence it achieves ideal consumption on only one case as can be seen in figure 4.1. Time complexity is lightly affected and doesn't have too much veriance as seen in figure 4.2.

4.3.3. Randomization. As seen in figure 4.1, randomization closely follows the baseline approach. For task set with higher utilization randomization provides power consumption of an average of 24% higher than the ideal consumption. Figure 4.2 shows us that time complexity for randomization doesn't change much as the number of tasks increases.

4.3.4. Non-Linear Programming. For NLP, the solution to our objective function is a $(1xn)$ integer vector where a 0 represents that the task is allocated on the LITTLE core and a 1 represents that it is allocated on the big core. From the figure we can observe

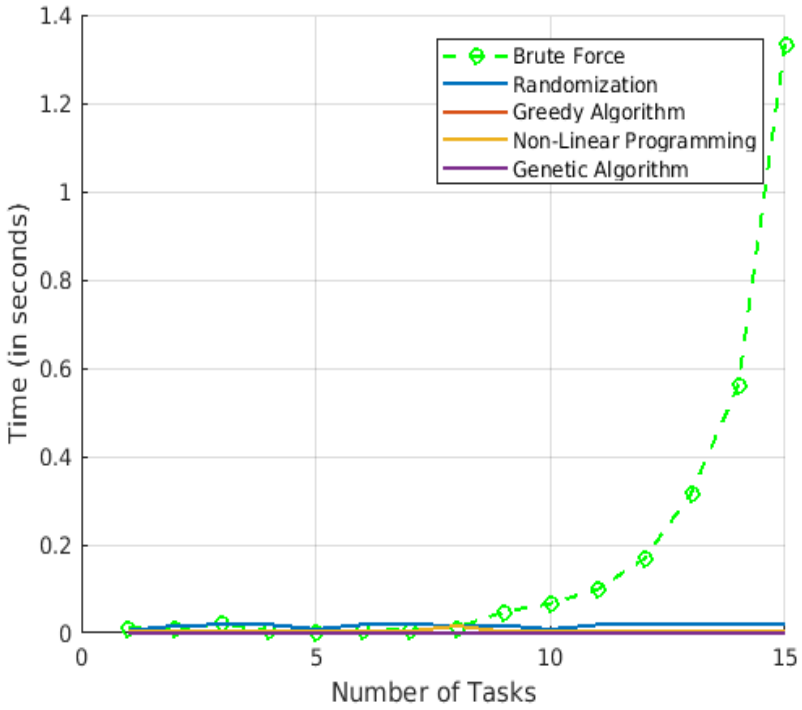

Figure 4.2. Time complexity of each proposed technique as the number of tasks increases

that NLP gives consistently bad results as seen in figure 4.1. The reason being that values obtained from NLP are fractional between 0 and 1. These values are then rounded off to the nearest integer in an attempt to map the nearest least power consumption. Time complexity stays within milliseconds as seen in figure 4.2.

4.3.5. Genetic Algorithm. As mentioned earlier, the minimum power consumed by genetic algorithm equal to the minimum power consumption observed with brute force method for majority of the cases. In figure 4.1, the orange line almost perfectly overlaps the blue line, which represent the power consumption due to genetic algorithm and the brute force method respectively. Figure 4.2 shows us that the time complexity for GA doesn't fluctuate much as the number of tasks increases, which is a desirable quality tasking into account the results for power consumption it provides.

4.4. CONCLUSION

For real-time systems timings constraints are an important factor to consider. We can conclude from the results that Genetic Algorithm provides us with optimal results within minimal time, which makes it desirable to incorporate in a real-time system. To take advantage of the heterogeneous architecture, we can have one of the cores actively running the task-to-core mapping technique giving us placements for the upcoming tasks.

5. ENERGY EFFICIENT REAL-TIME OPERATING SYSTEM

Implementing a truly energy-efficient system is a multi stage process. It can consist of not only having an energy efficient processor but also the operating system running over it. Thus far we have seen viable methods for segregating tasks on different types of processors that minimizes the cumulative energy consumption of all processors without violating utilization constrains. Following that we discovered an algorithm that further reduces energy consumption by considering the frequency/speed profile of every task and pairs them with the most suitable one by process of *task merging* to create a resultant speed profile that can be scheduled on the alloted processor. Until this point, implementation could be carried out assuming you have an underlying operating system that handles the actual scheduling of tasks through system calls and APIs. But what if the underlying operating system is not energy-efficient itself? The later part of this thesis focuses on the importance of having an energy efficient OS and explains the various stages of designing an OS where energy can be saved.

5.1. REAL-TIME OPERATING SYSTEM

An operating system is an extremely important system software that manages your computer's memory and processes, as well as its software and hardware. It is represented as a layer that sits between your applications and the hardware. Without an OS, applications would need to be coded to interact directly with the hardware, which would make the application inflexible and non-portable. Having an OS gives you the advantage of not having to worry about hardware specific details and lets you concentrate on building the application through use of common libraries. An OS can handle running multiple applications on the same hardware at the same time, providing a sense of multi-tasking to the user, hence most applications are programmed to be OS specific rather than hardware specific. The OS

is responsible for handling your hardware resources which includes input devices such as mouse and keyboard, output devices like printers or monitors, network components like routers or adapters as well as storage devices like external or internal drives. You can find an OS to be present almost all devices that consists of a computer like mobile phones, laptops, game consoles etc.

A Real-Time Operating System(RTOS) is an operating system with well defined timings constraints. Analogous to a real-ime task, an RTOS has the dual restriction of being logically as well as temporally correct, which means that the generated output not only has to be logically right but also has to be within the timing restriction given, else the system would fail. An RTOS typically processes data as it comes in without buffer delays and is widely used for applications with severe time bounds. Generally they are time sharing or event driven. A time sharing system uses system clock interrupts to switch between tasks while an event driven system utilizes task priorities for switching. The specific RTOS we consider in this thesis is called QP by Quantum Leaps, which will be ported on a TI C2000 based micro-processor.

5.2. QUANTUM PLATFORM - RTOS

Quantum Platform (QP) is a family of real-time framework offered by Quantum Leaps based on active objects for building embedded software. The family consists of QP/C, QP/C++ and QP-nano which are all open source, lightweight frameworks that can completely replace traditional RTOS on bare-metal single chip micro-controllers. The behavior of active object is adopted from Hierarchical State Machines (UML Statecharts). The framework gives us a selection of built-in real-time kernels (RTOS kernels) like the co-operative QV kernel, preemptive QK kernel or the dual-mode QXK kernel. The scope of this thesis is focused on the QP/C framework with the preemptive QK kernel.

Kernel. A kernel is often referred to as the heart of any OS. It has complete control over the memory, cpu, task scheduling and also responsible for task management, memory management, disk managements and process management as described by Webopedia Kernel Description. For any system there are limited resources that might be demanded by multiple applications. The kernel consists a layer of hardware abstraction that curtains the low level interface for connecting software with the hardware. The kernel decides when and how long a task gets to access a particular hardware resource while attempting to provide a fair share to all tasks while maintaining correctness. QP gives us the option to choose from three types of kernels, one being the co-operative QV kernel, also known as the Vanilla Kernel, second is the preemptive QK kernel and lastly a dual-mode QXK kernel which acts as a hybrid version of the first two.

Co-operative QV Kernel. The co-operative QV, or as previously known 'vanilla' kernel schedules active objects one at a time according to Quantum Leaps, QV kernel. It deploys a priority based algorithm that searches the ready queue for an active job with the highest priority and dispatches it to the related active object. As event processing duration for state machines are naturally short, the vanilla kernel is sufficient in most cases.

Preemptive QK Kernel. The preemptive QK kernel is designed such that it runs non-blocking active objects according to Quantum Leaps, Qk kernel. Active object managements is analogous to how an interrupt is handled using single stack by a Prioritized Interrupt Controller (like NVIC in Cortex-M). Here nesting of active objects is allowed where a higher priority object can preempt a lower priority one. Active objects are executed in a Run-to-Completion (RTC) fashion and are removed from the call stack upon completion, similar to how nested interrupts are removed from stack. This kernel follows the RM Schedule and can be used in hard real-time Applications.

Dual-mode QXK Kernel. The dual-mode QXK Kernel behaves exactly like a conventional RTOS described by Quantum Leaps, QXK kernel. It is a small, preemptive kernel that can execute basic threads like active objects along with traditional blocking

extended threads. It is specifically designed to mix traditionally blocking code along with event-driven active object execution. The scope of this work pertains to the preemptive QK kernel.

We will also encounter various steps during the port and execution of QP/C where modifications have been made keeping energy conservation in mind.

5.3. PROBLEM STATEMENT

Given an RTOS, we need to port it over to our desired platform making it energy-efficient such that it consumes minimal energy for functioning.

5.4. HARDWARE PLATFORM

Texas Instruments (TI) provides a wide range of embedded processors like ARM-based micro-controllers to Digital Signal Processors(DSP) to choose from based on requirements. 32-bit real-time C2000 micro-controller family from TI includes C2000 fixed-point, Piccolo, Delfino and Concerto Series. The platform we use is TMS320F28377S, a single core 32-bit floating-point micro-controller unit belonging to the Delfino SeriesTI TMS320F28377S Delfino. It consists of six GPIO ports with support for onboard flash memory up to 1MB and up to 164KB of SRAM. For more details regarding the platform please refer to TI TMS320F28377S Datasheet. It is ideal for closed loop applications such as digital power, servo motor control, solar inverters etc.

5.5. PORTING

Porting is the process of adapting a software to an environment it was not originally meant for. Often RTOSs are written with the idea of making it 'portable' which helps improve their market along with scalability. The system is coded in stages that separate the hardware dependent layer/code from the actual functioning of the OS itself. This makes

it easier to port an OS over different platforms by making changes only for the hardware specific features rather than having the whole application rewritten, eventually making it convenient and speeding up their release times.

Hardware specific changes are mainly incorporated in a Board Support Package (BSP) that is tuned specifically for the target platform. A typical BSP written in C would be saved as 'bsp.c'. Some other changes that are incorporated are the device specific header files, drivers and command files. Device specific header files and drivers define the variables, structures and methods the are typically used to interface with the device. It includes private details like structure of registers for the device, structures for defining the bits used in those registers as well as macros, constants and other define statements. Some public functions/methods like initializing the peripherals or accessing General Purpose Input-Output (GPIO) pins etc. can also be found in these files. They are generally stored under a '$device\_xx$' folder as '$device\_xx\_driver.c/.h$' files. Other files such as command or linker files provide methods to build with different configurations such as 'debug' or 'release' and also help map software code and data into hardware memory.

Porting QP/C to TMS320F28377S. Beginning from QP version 4.5.04 onwards, support for TMS320F28x along with a wide range of other platforms was dropped to reduce the release time for QP/C; these platforms are since identified as '*legacy platforms*'. We carry out porting of QP/C version 6.0.4 on TI C2000 based TMS320F28377S platform.

The basic philosophy of building embedded applications and the distribution of QP frameworks was changed since QP v5.4Quantum Leaps QP/C Revision History. Traditionally QP framework distribution and their port for supported platforms were independent, this release combines the QP baseline code with all available development kits to avoid any potential mistakes in downloading and installation of separate pieces of code. Additionally it also modifies the fundamental concept of building embedded application with the QP framework. All projects from then on include the QP framework as source code instead

of statically linking to libraries. Doing so maintained the correctness of compiler configurations and ensures consistent tool-set options are applied to application along with the framework code.

Project Structure. We modify the project structure of an old port for TMS320F28x to be consistent with all updates, as well as incorporate changes to consume minimal power during runtime. '*Dining philosopher*' example was structured using Code Composer Studio (CCSv7). The challenges and remedies carried out are as follows:

1. Initial step is to create separate folders for QP source and port files. It is important to create them as 'Linked folders'. This can be done by expanding the 'Advanced' tab in the window for adding a new folder and selecting 'Link to alternate location (Linked Folder)' option. Typically four folders need to be created:
   (a) QK: Linked to folder containing the source code for the preemptive Run-To-Completion (Non-blocking) QK kernel implementation.
   (b) QF: Linked to folder containing the source code of Active Object framework.
   (c) QP_include: Linked to folder containing the include header files for QP/C
   (d) QP_port: Linked to folder containing port files. We can choose to remove the debug, rel and spy folders and only let the header files reside in that folder.
2. Main application and board support package files are added to the project in the same way, i.e as linked files.
3. Update include path for any file that we would be adding. CCS would look for included files in the '*include paths*'. The successful building of project is conditional on the compiler finding those files at the specified path.
4. Current version of QP/C adopts the standard integer header file (stdint.h and stdbool.h) over the traditional non-standard *uint_t* data types for stricter type analysis. In qep_port.h, found in the QP_port linked folder, we include <stdbool.h> and <stdint.h> while commenting out all other defined data types except *uint8_t*.

Note: In rare cases CCS will throw an error stating it cannot find "stdbool.h" or "stdint.h". To fix it we need to make sure the compiler tool's include folder is added to the search path in CCS.

5. Many factors in C code are compiler dependent. A code that runs on one compiler does not guarantee the same expected execution on another compiler. Hence we need to make sure that the right compiler is chosen for our project. In our case we would choose Compiler Version TI v6.1.x under CCS General Settings.
6. Under General setting, we will choose our project belongs to C2000 family and select TMS320F28377S as the variant.
7. Configuration setting for the project need to be modified as well. Right-click on Project–>Properties and change the run-time support library to <automatic>. Make sure the correct command file is selected. The linker command file is responsible for mapping your code and data into memory. In our case, we would be using F2837x_FLASH_lnk.cmd that can usually be found coupled with device drivers. Under CCS Build options, check the box for 'Use default build command' in the Builder tab.
8. An additional command file would need to be passed to the linker based on whether our project is BIOS or non-BIOS based. This header linker command file is required to link the peripheral structures to the proper locations within the memory map. Since our project is non-BIOS based, we will add F2837xS_Header_nonBIOS.cmd in "Include command file" options for the linker. This file is also generally coupled with device drivers.

So far we have generated the required project directory structure and configuration setting for CCS. Here on we discuss about the specific features that can be modified for energy-efficient functioning of RTOS.

Modification for Energy Efficiency. As mentioned before, porting for a specific platform usually requires a BSP to be written for that platform. A *bsp.c* file pertains to all code and functions related to a particular hardware. It may include the selection and configuration of system clock and various other peripherals. It may also include the platform dependent functions like specifying an Interrupt Service Routine(ISR) in case the program is interrupted from a known source. A '*bsp_init()*' function is called from main that takes care of the hardware specific initialization.

1. A device header file is absolutely necessary and one of the very first things that should be included in the *bsp.c* file. The device header file defines the various typedefs for variables consistent with the device along with calling other platform dependent header files that manage the system's GPIO function, PIE Control, PIE Vector Tables etc.
2. Unbounded Input/Outputs in the system can cause leakage power consumptions, which in many cases is a major contributor to the overall power consumption in a micro-processor. Hence as soon as *bsp_init()* function is called, the first task is to call a system initialization function that disables the watchdog timer, enables Pull-Ups on unbounded IOs to reduce power consumption and also checks if the device is trimmed or not, apply static calibration values if it is not.
3. Initialize the Phase Locked Loop(PLL) control and manage the peripheral clocks. Here we selectively turn off clocks for all peripherals except for ones that are actually used in the application. For our project we require Timer0, I2C, PWM and ADCA, thus turn on clocks for only the related peripherals and consume lesser static energy due to unused peripheral components.
4. Initialize the Peripheral Interrupt Enable (PIE) Control and map the PIE Vector Table to their designated ISRs. We direct all PIE Vector addresses to point to an illegal ISR except for the peripherals we actually use. An illegal ISR is nothing but an infinite

loop which would help us detect an unexpected or illegal execution in the application. As mentioned before, our application uses Timer0, PWM, I2C and ADCA thus when an interrupt occurs from these known sources, control is transferred to their respective ISRs. It is a well known fact that RAM memory provides the quickest access and search results compared to all other memories available. Taking advantage of this fact, the required ISRs have been placed in the RAM memory for greater speed and efficiency. Thus our application spends lesser clock cycles and hence spends less energy in searching for data or code to execute.

5. QP uses the Timer0 to clock its "ticks" called *QK_tick*. These can be essentially understood as the heartbeats of QP which takes its decisions once every *QK_tick*. Every time an interrupt is thrown by Timer0, its ISR calls a scheduler which selects the active object with the highest priority with a non-empty event queue and dispatches it for execution. To integrate PLECS with QP, we change the Timer0's ISR such that it allows interrupt from only one peripheral, the ADCA, to preempt QP. The status of all peripheral registers before entering the main Timer0 ISR is saved and all interrupts except the ADCA interrupt are disabled. A counter keeps a track of the interrupt nesting level. The original peripheral interrupt status is restored after the ISR is completed. This makes sure that processor is always available for data conversion even if QP temporarily disables all other interrupts. Also saving and restoring the interrupt status allows us to maintain the integrity of the system.

6. Lastly a great chunk of energy savings is achieved by putting the processor on *IDLE* mode for majority of the execution cycle. IDLE mode is essentially a "Low-Power" mode where the CPU1 is turned to IDLE mode and flash is powered down, the processor can be woken up from the idle mode by any of the interrupts. Thus the processor spends most of its life consuming around 80 mA in idle mode whereas it consumes 250 mA in normal operational mode. This way the processor is only active when there is an interrupt required to be served, else can remain idle.

To sum up, we have created a custom version of a Real-Time Operating System, QP, that uses active objects based on Hierarchical State Machines(UML Statecharts). The created version provides efficient energy savings by utilizing minimal peripheral restricted to requirements of our application. The RTOS is modified to save on static energy consumption by running the core on IDLE mode throughout most of its execution life and would be switched to operational mode only in case an event becomes available to process, this enables us to reduce energy consumption by almost 68%. Occasionally QP disabled all interrupts globally to serve Timer0 ISR, contrary to our custom version which allows interrupt through one peripheral, ADCA, at all times and keep the processor available even if all other interrupts are disabled globally. Even though our version of energy-efficient RTOS restricts only one interrupt to be available at all times, certainly the concept explained is reusable to add more interrupts based on application needs.

6. CONCLUSIONS

This thesis reports 3 major contribution to efficiently design a real-time system. The works can be understood as a multi-stage process, steps are provided to create an energy-efficient RTOS that consumes minimal itself over which applications are scheduled. Secondly different techniques are introduced that can efficiently partition a given real-time task set into different sets for power-aware execution on a HMP platform. Lastly, we introduce an algorithm that merges two tasks with similar speed-profiles and creates a resultant speed-profile for simultaneous execution of the two tasks on the same cluster, this further provides more energy savings. Energy is an extremely important resource that is carefully considered at every stage in this presented work. Results and simulations provided strongly back the given theory.

REFERENCES

Aydin, H. and Yang, Q., 'Energy-aware partitioning for multiprocessor real-time systems,' in 'Proceedings International Parallel and Distributed Processing Symposium,' ISSN 1530-2075, 2003 pp. 9 pp.–, doi:10.1109/IPDPS.2003.1213225.

Baker, T. P., 'A stack-based resource allocation policy for realtime processes,' in '[1990] Proceedings 11th Real-Time Systems Symposium,' 1990 pp. 191–200, doi:10.1109/REAL.1990.128747.

Bambagini, M., Marinoni, M., Aydin, H., and Buttazzo, G., 'Energy-aware scheduling for real-time systems: A survey,' ACM Trans. Embed. Comput. Syst., 2016, **15**(1), pp. 7:1–7:34, ISSN 1539-9087, doi:10.1145/2808231.

Chen, G., Huang, K., and Knoll, A., 'Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination,' ACM Trans. Embed. Comput. Syst., 2014, **13**(3s), pp. 111:1–111:21, ISSN 1539-9087, doi:10.1145/2567935.

Chen, J.-J. and Kuo, C.-F., 'Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms,' in 'RTCSA'07,' IEEE, 2007 .

Chen, J. J., Schranzhofer, A., and Thiele, L., 'Energy minimization for periodic real-time tasks on heterogeneous processing units,' in '2009 IEEE International Symposium on Parallel Distributed Processing,' ISSN 1530-2075, 2009 pp. 1–12, doi:10.1109/IPDPS.2009.5161024.

Chen, M.-I. and Lin, K.-J., 'Dynamic priority ceilings: A concurrency control protocol for real-time systems,' Real-Time Systems, 1990, **2**(4), pp. 325–346, ISSN 1573-1383, doi:10.1007/BF01995676.

Colin, A., Kandhalu, A., and Rajkumar, R., 'Energy-efficient allocation of real-time applications onto heterogeneous processors,' in '2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications,' ISSN 2325-1271, 2014 pp. 1–10, doi:10.1109/RTCSA.2014.6910506.

Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J.-M., and Wagner, F., 'Random graph generation for scheduling simulations,' in 'ICST'10,' 2010 .

Ellis, C. S., 'The case for higher-level power management,' in 'Proceedings of the Seventh Workshop on Hot Topics in Operating Systems,' 1999 pp. 162–167, doi:10.1109/HOTOS.1999.798394.

Energy Monitoring, 2017, https://github.com/tuxamito/emoxu3.

Feitelson, D. G. and Rudolph, L., 'Gang scheduling performance benefits for fine-grain synchronization,' Journal of Parallel and Distributed Computing, 1992, **16**(4), pp. 306 – 318, ISSN 0743-7315, doi:https://doi.org/10.1016/0743-7315(92)90014-E.

Gamma distribution, 2017, http://en.wikipedia.org/wiki/Gamma distribution.

Goodenough, J. B. and Sha, L., 'The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks,' in 'Proceedings of the Second International Workshop on Real-time Ada Issues,' IRTAW '88, ACM, New York, NY, USA, ISBN 0-89791-295-0, 1988 pp. 20–31, doi:10.1145/58612.59371.

Guo, Z., Bhuiyan, A., Saifullah, A., Guan, N., and Xiong, H., 'Energy-efficient multi-core scheduling for real-time dag tasks,' in 'LIPIcs-Leibniz International Proceedings in Informatics,' volume 76, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017a .

Guo, Z., Bhuiyan, A., Saifullah, A., Guan, N., and Xiong, H., 'Energy-efficient multi-core scheduling for real-time dag tasks,' in 'LIPIcs-Leibniz International Proceedings in Informatics,' Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017b .

Howard, J., Dighe, S., Vangal, S. R., Ruhl, G., Borkar, N., Jain, S., Erraguntla, V., Konow, M., et al., 'A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling,' IEEE Journal of Solid-State Circuits, ????

Huang, P., Kumar, P., Giannopoulou, G., and Thiele, L., 'Energy efficient dvfs scheduling for mixed-criticality systems,' in '2014 International Conference on Embedded Software (EMSOFT),' 2014 pp. 1–10, doi:10.1145/2656045.2656057.

Kong, F., Guan, N., Deng, Q., and Yi, W., 'Energy-efficient scheduling for parallel real-time tasks based on level-packing,' in 'Proceedings of the 2011 ACM Symposium on Applied Computing,' SAC '11, ACM, New York, NY, USA, ISBN 978-1-4503-0113-8, 2011 pp. 635–640, doi:10.1145/1982185.1982326.

Lee, D. C., Crowley, P. J., Baer, J.-L., Anderson, T. E., and Bershad, B. N., 'Execution characteristics of desktop applications on windows nt,' in 'Proceedings of the 25th Annual International Symposium on Computer Architecture,' ISCA '98, IEEE Computer Society, Washington, DC, USA, ISBN 0-8186-8491-7, 1998 pp. 27–38, doi:10.1145/279358.279366.

Li, K., 'Energy efficient scheduling of parallel tasks onmultiprocessor computers,' The Journal of Supercomputing, 2012, **60**(2), pp. 223–247, ISSN 1573-0484, doi:10.1007/s11227-010-0416-0.

Liu, C., Li, J., Huang, W., Rubio, J., Speight, E., and Lin, F. X., 'Power-efficient time-sensitive mapping in heterogeneous systems,' in '2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT),' 2012 pp. 23–32.

Liu, D., Spasic, J., Chen, G., and Stefanov, T., 'Energy-efficient mapping of real-time streaming applications on cluster heterogeneous mpsocs,' in '2015 13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia),' 2015 pp. 1–10, doi:10.1109/ESTIMedia.2015.7351764.

Lorch, J. R. and Smith, A. J., 'Scheduling techniques for reducing processor energy use in macos,' Wirel. Netw., 1997, **3**(5), pp. 311–324, ISSN 1022-0038, doi:10.1023/A:1019177822227.

Lorch, J. R. and Smith, A. J., 'Software strategies for portable computer energy management,' IEEE Personal Communications, 1998, **5**(3), pp. 60–73, ISSN 1070-9916, doi:10.1109/98.683740.

Narayana, S., Huang, P., Giannopoulou, G., Thiele, L., and Prasad, R. V., 'Exploring energy saving for mixed-criticality systems on multi-cores,' in '2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),' 2016 pp. 1–12, doi:10.1109/RTAS.2016.7461336.

Pagani, S. and Chen, J.-J., 'Energy efficient task partitioning based on the single frequency approximation scheme,' 2013, pp. 308–318.

Pagani, S. and Chen, J.-J., 'Energy efficiency analysis for the single frequency approximation (sfa) scheme,' ACM Trans. Embed. Comput. Syst., 2014, **13**(5s), pp. 158:1–158:25, ISSN 1539-9087, doi:10.1145/2660490.

Paolillo, A., Goossens, J., Hettiarachchi, P. M., and Fisher, N., 'Power minimization for parallel real-time systems with malleable jobs and homogeneous frequencies,' in '2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications,' ISSN 2325-1271, 2014 pp. 1–10, doi:10.1109/RTCSA.2014.6910538.

Qi, X. and Zhu, D.-K., 'Energy efficient block-partitioned multicore processors for parallel applications,' 2011, **26**, pp. 418–433.

Quantum Leaps, Qk kernel, 2017, `https://www.state-machine.com/qpc/group__qk.html`.

Quantum Leaps QP/C Revision History, 2017, https://www.state-machine.com/qpc/history.html.

Quantum Leaps, QV kernel, 2017, `https://www.state-machine.com/qpc/group__qv.html`.

Quantum Leaps, QXK kernel, 2017, `https://www.state-machine.com/qpc/group__qxk.html`.

Rt-App, 2017, https://github.com/scheduler-tools/rt-app/.

Saifullah, A., Ferry, D., Li, J., Agrawal, K., Lu, C., and Gill, C. D., 'Parallel real-time scheduling of dags,' IEEE Transactions on Parallel and Distributed Systems, 2014a, **25**(12), pp. 3242–3252, ISSN 1045-9219, doi:10.1109/TPDS.2013.2297919.

Saifullah, A., Ferry, D., Li, J., Agrawal, K., Lu, C., and Gill, C. D., 'Parallel real-time scheduling of dags,' IEEE TPDS, 2014b.

Sha, L., Rajkumar, R., and Lehoczky, J. P., 'Priority inheritance protocols: an approach to real-time synchronization,' IEEE Transactions on Computers, 1990, **39**(9), pp. 1175–1185, ISSN 0018-9340, doi:10.1109/12.57058.

TI TMS320F28377S Datasheet, 2016, http://www.ti.com/lit/ug/spruhx5e/spruhx5e.pdf.

TI TMS320F28377S Delfino, 2010, http://www.ti.com/product/TMS320F28377S?keyMatch=tms320f28377s&tisearch=Search-EN-Everything.

Vahdat, A., Lebeck, A., and Ellis, C. S., 'Every joule is precious: The case for revisiting operating system design for energy efficiency,' in 'Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System,' EW 9, ACM, New York, NY, USA, 2000 pp. 31–36, doi:10.1145/566726.566735.

Webopedia Kernel Description, 2010, https://www.webopedia.com/TERM/K/kernel.html.

Zhu, D., AbouGhazaleh, N., Mosse, D., and Melhem, R., 'Power aware scheduling for and/or graphs in multiprocessor real-time systems,' in 'Proceedings International Conference on Parallel Processing,' ISSN 0190-3918, 2002 pp. 593–601, doi:10.1109/ICPP.2002.1040917.

Zhu, D., Mosse, D., and Melhem, R., 'Power-aware scheduling for and/or graphs in real-time systems,' IEEE Trans. Parallel Distrib. Syst., 2004, **15**(9), pp. 849–864, ISSN 1045-9219, doi:10.1109/TPDS.2004.45.

VITA

The author, Aamir Khan, was born in Mumbai, India. He developed passion for technology early in life and received his Bachelor of Engineering in Electronics from Mumbai University, May 2015. After his bachelors he enrolled in Missouri University of Science and Technology for graduate studies. During his time at Missouri S&T, the author taught laboratory as well as aided in designing assignments for Real-Time Systems course under the guidance of Dr. Zhishan Guo at Missouri S&T.

The author started working on his master's degree in late 2016 and conducted research for the latter year of his duration in Energy Efficient Real-Time Systems. He received his Master's of Science in Computer Engineering from Missouri University of Science and Technology - Rolla, MO, USA, July 2018.